

20030210027

Best Available Copy

DTIC FILE COPY

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

(2)

AD-A233 134

of to average 1 hour per response, including the time for reviewing instructions, searching existing data sources (including the Internet), gathering and reviewing the data, and completing the review and report. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project Director (0704-0188), Washington, DC 20503.

1. REPORT DATE

2. REPORT TYPE AND DATES COVERED

Final 16 Jan 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report:

Verdix Corporation, VAda-110-6161, Version 6.0.2, DECstation 3100 (Host) to DECstation 3100 (Target), 900228W1.11001

5. FUNDING NUMBERS

6. AUTHOR(S)

Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB
Dayton, OH 454338. PERFORMING ORGANIZATION
REPORT NUMBER

AVF-VSR-363.0191

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-308110. SPONSORING/MONITORING AGENCY
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Verdix Corporation, VAda-110-6161, Version 6.0.2, Wright-Patterson AFB, OH, DECstation 3100 (Host to Target), ACVC 1.11.

DTIC
ELECTE
MAR 22 1991
S B D

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED18. SECURITY CLASSIFICATION
OF THIS PAGE
UNCLASSIFIED19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 28 February 1990.

Compiler Name and Version: VAda-110-6161, Version 6.0.2

Host Computer System: DECstation 3100, ULTRIX 3.1

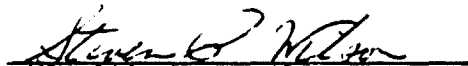
Target Computer System: DECstation 3100, ULTRIX 3.1

Customer Agreement Number: 90-01-15-VRX


See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 900228W1.11001 is awarded to Verdix Corporation. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.


Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


Ada Joint Program Office
Dr. John Solomon, Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



AVF Control Number: AVF-VSR-363.0191
16 January 91
90-01-15-VRX

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900228W1.11001
Verdix Corporation
VAda-110-6161, Version 6.0.2
DECstation 3100 => DECstation 31C0

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer: Verdex Corporation
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
ACVC Version: 1.11
Ada Implementation:
Compiler Name and Version: VAda-110-6161, Version 6.0.2
Host Computer System: DECstation 3100, ULTRIX 3.1
Target Computer System: DECstation 3100, ULTRIX 3.1

Customer's Declaration

I, the undersigned, representing Verdex Corporation, declare that Verdex Corporation has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that the Verdex Corporation is the owner of the above implementation and the certificates shall be awarded in the name of the owner's corporate name.

Stephen Scalpone
Stephen Zeigler *SCALPONE*
Verdex Corporation
1600 NW Compton Drive
Suite 357
Beaverton, Oregon 97006

Date: 2/27/90

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES	1-2
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS	2-1
2.3	TEST MODIFICATIONS	2-3
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

INTRODUCTION

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.
- [JG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of Identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

INTRODUCTION

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 31 January 1990.

E28005C	B41308B	C45114A	C45612B	C45651A	C46022A
B49008A	A74006A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	C97116A	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	BD8002A
BD8004C	CD9005A	CD9005B	CDA201E	CE2107I	CE2119B
CE3111C	CE3118A	CE3411B	CE3412B	CE3812A	CE3902B

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

IMPLEMENTATION DEPENDENCIES

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type `LONG_INTEGER`:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35702B, C35713C, B86001U, and C86006G check for the predefined type `LONG_FLOAT`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation that range exceeds the safe numbers and must be rejected. (See section 2.3)

C45346A checks that `NUMERIC_ERROR` is not raised if the length of the result of catenation exceeds `INTEGER'LAST` or `SYSTEM.MAX_INT` and that `CONSTRAINT_ERROR` or `STORAGE_ERROR` is raised instead. `NUMERIC/CONSTRAINT_ERROR` was raised when an array of length `INTEGER'LAST/2 + 1` was declared.

C45531M..P (4 tests) and C45532M..P (4 tests) use a value for `SYSTEM.MAX_MANTISSA` of 47 or greater.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C86001F recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete. For this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range of `DURATION`. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

IMPLEMENTATION DEPENDENCIES

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN FILE	SEQUENTIAL IO
CE2102E	CREATE	OUT FILE	SEQUENTIAL IO
CE2102F	CREATE	INOUT FILE	DIRECT IO
CE2102I	CREATE	IN FILE	DIRECT IO
CE2102J	CREATE	OUT FILE	DIRECT IO
CE2102N	OPEN	IN FILE	SEQUENTIAL IO
CE2102O	RESET	IN FILE	SEQUENTIAL IO
CE2102P	OPEN	OUT FILE	SEQUENTIAL IO
CE2102Q	RESET	OUT FILE	SEQUENTIAL IO
CE2102R	OPEN	INOUT FILE	DIRECT IO
CE2102S	RESET	INOUT FILE	DIRECT IO
CE2102T	OPEN	IN FILE	DIRECT IO
CE2102U	RESET	IN FILE	DIRECT IO
CE2102V	OPEN	OUT FILE	DIRECT IO
CE2102W	RESET	OUT FILE	DIRECT IO
CE3102E	CREATE	IN FILE	TEXT IO
CE3102F	RESET	Any Mode	TEXT IO
CE3102G	DELETE	-----	TEXT IO
CE3102I	CREATE	OUT FILE	TEXT IO
CE3102J	OPEN	IN FILE	TEXT IO
CE3102K	OPEN	OUT FILE	TEXT IO

CE2203A and CE2403A check that WRITE raises USE ERROR if the capacity of the external file is exceeded, for SEQUENTIAL IO and DIRECT IO respectively. This implementation cannot restrict file capacity.

CE3304A checks that USE ERROR is raised if a call to SET LINE LENGTH or SET PAGE LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 22 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

IMPLEMENTATION DEPENDENCIES

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B85008G	B85008H	BC1303F	BC3005B	BD2B03A	BD2D03A
BD4003A					

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO; the compiler rejects the use of the range `FLOAT'FIRST..FLOAT'LAST` as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. ARM 3.5.7(12)).

CD1009A, CD1009I, CD1C03A, CD2A22J, CD2A24A, and CD2A31A..C (3 tests) use instantiations of the support procedure `Length_Check`, which uses `Unchecked_Conversion` according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instantiations of `Length_Check`--i.e., the allowed `Report.Failed` messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Steve Hodges
Verdix Corporation
14130-A Sully Field Circle
Chantilly, VA 22021

For a point of contact for sales information about this Ada implementation system, see:

Steve Hodges
Verdix Corporation
14130-A Sully Field Circle
Chantilly, VA 22021

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

PROCESSING INFORMATION

a) Total Number of Applicable Tests	3836	
b) Total Number of Withdrawn Tests	60	
c) Processed Inapplicable Tests	73	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	274	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 274 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were not loaded directly onto the host computer. The tape was loaded onto a Sun Workstation and the tests were copied over Ethernet to the host machine.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

-v Suppress generation of warning messages.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "69.0E1"
\$BIG_STRING1	'" & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'" & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'" & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$MAX_IN_LEN	499
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2_147_483_647
\$DEFAULT_MEM_SIZE	16_777_216
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	DEC_RISC
\$DELTA_DOC	0.0000000004656612873077392578125
\$ENTRY_ADDRESS	SYSTEM."+"(16#A#)
\$ENTRY_ADDRESS1	SYSTEM."+"(16#B#)
\$ENTRY_ADDRESS2	SYSTEM."+"(16#C#)
\$FIELD_LAST	2_147_483_647
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_TYPE
\$FLOAT_NAME	NO_SUCH_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	100_000.0
\$GREATER_THAN_DURATION BASE LAST	10_000_000.0
\$GREATER_THAN_FLOAT_BASE LAST	1.8E+308
\$GREATER_THAN_FLOAT_SAFE LARGE	5.0E307
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	

MACRO PARAMETERS

```

                                9.0E37

$HIGH_PRIORITY                99

$ILLEGAL_EXTERNAL_FILE_NAME1  "/illegal/file_name/2}]X2102C.DAT"

$ILLEGAL_EXTERNAL_FILE_NAME2  "/illegal/file_name/CE2102C*.DAT"

$INAPPROPRIATE_LINE_LENGTH    -1

$INAPPROPRIATE_PAGE_LENGTH    -1

$INCLUDE_PRAGMA1              PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2              PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST                -2_147_483_648

$INTEGER_LAST                  2_147_483_647

$INTEGER_LAST_PLUS_1          2_147_483_648

$INTERFACE_LANGUAGE           C

$LESS_THAN_DURATION           -100_000.0

$LESS_THAN_DURATION_BASE_FIRST -10_000_000.0

$LINE_TERMINATOR              ASCII.LF & ASCII.FF

$LOW_PRIORITY                  0

$MACHINE_CODE_STATEMENT       CODE_0'(OP => NOP);

$MACHINE_CODE_TYPE            CODE_0

$MANTISSA_DOC                  31

$MAX_DIGITS                    15

$MAX_INT                       2_147_483_647

$MAX_INT_PLUS_1                2_147_483_648

$MIN_INT                       -2_147_483_648

$NAME                          TINY_INTEGER

```

MACRO PARAMETERS

\$NAME_LIST	DEC_RISC
\$NAME_SPECIFICATION1	/usr/validation/tests/c/e/X2120A
\$NAME_SPECIFICATION2	/usr/validation/tests/c/e/X2120B
\$NAME_SPECIFICATION3	/usr/validation/tests/c/e/X3119A
\$NEG_BASED_INT	16#FFFFFFFD#
\$NEW_MEM_SIZE	16_777_216
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	DEC_RISC
\$PAGE_TERMINATOR	ASCII.FF
\$RECORD_DEFINITION	RECORD SUBP: OPERAND; END RECORD;
\$RECORD_NAME	CODE_0
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	1024
\$TICK	0.01
\$VARIABLE_ADDRESS	VAR_1'ADDRESS
\$VARIABLE_ADDRESS1	VAR_2'ADDRESS
\$VARIABLE_ADDRESS2	VAR_3'ADDRESS
\$YOUR_PRAGMA	PASSIVE

APPENDIX B
COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

ada

ada -- Ada compiler

Syntax

ada [options] [source_file]... [linker_options] [object_file.o]...

Options

- | | |
|---------------------------------|--|
| -# <i>identifier type value</i> | (define) Define an identifier of a specified type and value. See Chapter 10, <i>VADS ADA PREPROCESSOR REFERENCE</i> . |
| -a <i>file_name</i> | (archive) Treat <i>file_name</i> as an object archive file created by ar. Since some archive files end with .a, -a is used to distinguish archive files from Ada source files. |
| -DG | Write out the gnrx.lib file in ASCII. |
| -d | (dependencies) Analyze for dependencies only. Do not do semantic analysis or code generation. Update the library, marking any defined units as uncompiled. The -d option is used by a.make to establish dependencies among new files. |
| -e | (error) Process compilation error messages using a.error and send it to standard output. Only the source lines containing errors are listed. Only one -e or -E option should be used. |
| -E | |
| -E <i>file</i> | |
| -E <i>directory</i> | (error output) Without a file or directory argument, ada processes error messages using a.error and directs a brief output to standard output; the raw error messages are left in ada_source.err. If a file pathname is given, the raw error messages are placed in that file. If a directory argument is supplied, the raw error output is placed in dir/source.err. The file of raw error messages can be used as input to a.error. Only one -e or -E option should be used. |
| -el | (error listing) Intersperse error messages among source lines and direct to standard output. |
| -E | |
| -E <i>file</i> | |
| -E <i>directory</i> | (error listing) Same as the -E option, except that source listing with errors is produced. |
| -ev | (error vi(1)) Process syntax error messages using a.error, embed them in the source file, and call the environment editor ERROR_EDITOR. (If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN should also be defined. ERROR_PATTERN is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specified, vi(1) is invoked. |
| -K | (keep) Keep the intermediate language (IL) file produced by the compiler from end. The IL file will be placed in the .objects directory, with the file name Ada_source.i |
| -L <i>library_name</i> | (library) Operate in VADS library <i>library_name</i> (the current working directory is the default). |
| -file <i>abbreviation</i> | (library search) This is an option passed to the UNIX linker, ld(1) telling it to search the specified library file. (No space between the -l and the file abbreviation.) |

For a description of the file abbreviations, see also Operating system documentation, Id(1).

-M *unit_name*

(main) Produce an executable program by linking the named unit as the main program. *unit_name* must already be compiled. It must be either a parameterless procedure or a parameterless function returning an integer. The executable program will be named *a.out*; unless overridden with the **-o** option.

-M *source_file*

(main) Produce an executable program by compiling and linking *source_file*. The main unit of the program is assumed to be the root name of the *.a* file (for *foo.a* the unit is *foo*). Only one *.a* file may be preceded by **-M**. The executable program will be named *a.out* unless overridden with the **-o** option.

-o *executable_file*

(output) This option is to be used in conjunction with the **-M** option. *executable_file* is the name of the executable rather than the default *a.out*.

-O[0-9]

(optimize) Invoke the code optimizer (OPTIM3). An optional digit (there is no space before the digit) provides the level of optimization. The default is **-O4**.

-O full optimization

-O0 prevents optimization

-O1 no hoisting

-O2 no hoisting but more passes

-O3 no hoisting but even more passes

-O4 hoisting from loops

-O5 hoisting from loops but more passes

-O6 hoisting from loops with maximum passes

-O7 hoisting from loops and branches

-O8 hoisting from loops and branches, more passes

-O9 hoisting from loops and branches, maximum passes

Hoisting from branches (and cases alternatives) can be slow and does not always provide significant performance gains so it can be suppressed.

For more information about optimization, see *COMPILING ADA PROGRAMS*, *Compiler Optimizations* on page UG 4-3. See also pragma *OPTIMIZE_CODE(OFF)* on page PG F-4.

-P

Invoke the Ada Preprocessor. See Chapter 10, *VADS ADA PREPROCESSOR REFERENCE*.

-R *VADS_library*

(recompile instantiation) Force analysis of all generic instantiations, causing reinstantiation of any that are out of date.

-S

(suppress) Apply pragma *SUPPRESS* to the entire compilation for all suppressible checks. (See also pragma *SUPPRESS(ALL_CHECKS)* on page PG F-5)

-T

(timing) Print timing information for the compilation.

ada

- v** (verbose) Print compiler version number, date and time of compilation, name of file compiled, command input line, total compilation time, and error summary line. Storage usage information about the object file is provided. With OPTIM3 the output format of compression (the size of optimized instructions) is as a percentage of input (unoptimized instructions).
- w** (warnings) Suppress warning diagnostics.

Description

The command **ada** executes the Ada compiler and compiles the named Ada source file, ending with the **.a** suffix. The file must reside in a VADS library directory. The **ada.lib** file in this directory is modified after each Ada unit is compiled.

By default, **ada** produces only object and net files. If the **-M** option is used, the compiler automatically invokes **a.ld** and builds a complete program with the named library unit as the main program.

Non-Ada object files (**.o** files produced by a compiler for another language) may be given as arguments to **ada**. These files will be passed on to the linker and will be linked with the specified Ada object files.

Command line options may be specified in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

Several VADS compilers may be simultaneously available on a single system. Because the **ada** command in any **VADS_location/bin** on a system will execute the correct compiler components based upon visible library directives, the option **-sh** is provided to print the name of the components actually executed.

Program listings with a disassembly of machine code instructions are generated by **a.db** or **a.das**.

See also **a.das** on page UG 8-6; **a.db** on page UG 8-7; **a.error** on page UG 8-10; **a.ld** on page UG 8-14; **a.mklib** on page UG 8-21; and Operating System reference documentation for the **ld(1)** utility.

Diagnostics

The diagnostics produced by the VADS compiler are intended to be self-explanatory. Most refer to the RM. Each RM reference includes a section number and optionally, a paragraph number enclosed in parentheses.

COMPILATION SYSTEM OPTIONS

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

a.ld — prelinker

Syntax

a.ld [options] unit_name [ld_options]

Description

a.ld collects the object files needed to make *unit_name* a main program and calls the UNIX linker ld(1) to link together all Ada and other language objects required to produce an executable image in a.out. *unit_name* is the main program and must name a non-generic subprogram. If *unit_name* is a function, it must return a value of the type STANDARD.INTEGER. This integer result will be passed back to the UNIX shell as the status code of the execution. The utility uses the net files produced by the Ada compiler to check dependency information. a.ld produces an exception mapping table and a unit elaboration table and passes this information to the linker.

a.ld reads instructions for generating executables from the ada.lib file in the VADS libraries on the search list. Besides information generated by the compiler, these directives also include WITHn directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITHn directives may be placed into a library, but they must be numbered contiguously beginning at WITH1. The directives are recorded in the library's ada.lib file and have the following form.

```
WITH1:LINK:object_file:
WITH2:LINK:archive_file:
```

WITHn directives may be placed in the local Ada libraries or in any VADS library on the search list.

A WITHn directive in a local VADS library or earlier on the library search list will hide the same numbered WITHn directive in a library later in the library search list.

Use the tool a.info to change or report library directives in the current library.

All arguments after *unit_name* are passed on to the linker. These may be options for it, archive libraries, library abbreviations, or object files.

VADS_location/bin/a.ld is a wrapper program that executes the correct executable based upon directives visible in the ada.lib file. This permits multiple VADS compilers to exist on the same host. The -sh option prints the name of the actual executable file.

Options

- DX (debug) Debug memory overflow (use in cases where linking a large number of units causes the error message "local symbol overflow" to occur).
- E unit_name (elaborate) Elaborate *unit_name* as early in the elaboration order as possible.
- F (files) Print a list of dependent files in order and suppress linking.
- L library_name (library) Operate in VADS library *library_name* (the current working directory is the default).
- o executable_file (output) Use the specified file name as the name of the output rather than the default, a.out.
- sh (show) Display the name of the tool executable but do not execute it.

a.ld

- U** (units) Print a list of dependent units in order and suppress linking.
- v** (verbose) Print the linker command before executing it.
- V** (verify) Print the linker command but suppress execution.

Files

- a.out** default output file
- .nets** Ada DIANA net files directory
- .objects/*** Ada object files
- VADS_location/standard/*** startup and standard library routines

See also *Operating system documentation*, ld(1).

Diagnostics

Self-explanatory diagnostics are produced for missing files, etc. Occasional additional messages are produced by the linker.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

```
...
type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type TINY_INTEGER is range -128 .. 127;

type FLOAT is digits 15 range
    -16#0.1FFFFFFFFFFF# .. 16#0.1FFFFFFFFFFF#;
type SHORT_FLOAT is digits 6 range -16#0.FFFFFF# .. 16#0.FFFFFF#;

type DURATION is delta 0.001 range -2147483.648 .. 2147483.647;
...
```

end STANDARD;

ATTACHMENT I

APPENDIX F. Implementation-Dependent Characteristics

1. Implementation-Dependent Pragas

1.1. `INLINE_ONLY` Pragma

The `INLINE_ONLY` pragma, when used in the same way as pragma `INLINE`, indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space. If a user erroneously makes an `INLINE_ONLY` subprogram recursive a warning message will be emitted and an `PROGRAM_ERROR` will be raised at run time.

1.2. `BUILT_IN` Pragma

The `BUILT_IN` pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the `MACHINE_CODE` package.

1.3. `SHARE_CODE` Pragma

The `SHARE_CODE` pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers `TRUE` or `FALSE` as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is `TRUE` the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is `FALSE` each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma `SHARE_BODY` is also recognized by the implementation and has the same effect as `SHARE_CODE`. It is included for compatibility with earlier versions of VADS.

1.4. `NO_IMAGE` Pragma

The pragma suppresses the generation of the image array used for the `IMAGE` attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. An attempt to use the `IMAGE` attribute on a type whose image array has been suppressed will result in a compilation warning and `PROGRAM_ERROR` raised at run time.

1.5. `EXTERNAL_NAME` Pragma

The `EXTERNAL_NAME` pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

1.6. INTERFACE_NAME Pragma

The `INTERFACE_NAME` pragma takes the name of a variable or subprogram defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable or subprogram name with an external reference to the second, `link_argument`. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object or subprogram declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following:

- a loop variable,
- a constant,
- an initialized variable,
- an array, or
- a record.

1.7. IMPLICIT_CODE Pragma

Takes one of the identifiers `ON` or `OFF` as the single argument. This pragma is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if `OFF` is used and any implicit code needs to be generated. The default is `ON`.

1.8. OPTIMIZE_CODE Pragma

Takes one of the identifiers `ON` or `OFF` as the single argument. This pragma is only allowed within a machine code procedure. It specifies whether the code should be optimized by the compiler. The default is `ON`. When `OFF` is specified, the compiler will generate the code as specified.

2. Implementation of Predefined Pragmas

2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type `ADDRESS` in `SYSTEM`. All parameters must have mode `IN`. Record and array objects can be passed by reference using the `ADDRESS` attribute.

2.5. LINK_WITH

Can be used to pass arguments to the target linker. It may appear in any declarative part and accepts one argument, a constant string expression. This argument is passed to the target linker whenever the unit containing the pragma is included in a link.

2.6. LIST

This pragma is implemented as described in Appendix B of the Ada RM.

2.7. MEMORY_SIZE

This pragma is recognized by the implementation. The implementation does not allow `SYSTEM` to be modified by means of pragmas, the `SYSTEM` package must be recompiled.

2.8. NON_REENTRANT

This pragma takes one argument which can be the name of either a library subprogram or a subprogram declared immediately within a library package spec or body. It indicates to the compiler that the subprogram will not be called recursively allowing the compiler to perform specific optimizations. The pragma can be applied to a subprogram or a set of overloaded subprograms within a package spec or package body.

2.9. NOT_ELABORATED

This pragma can only appear in a library package specification. It indicates that the package will not be elaborated because it is either part of the RTS, a configuration package or an Ada package that is referenced from a language other than Ada. The presence of this pragma suppresses the generation of elaboration code and issues warnings if elaboration code is required.

2.10. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

2.11. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. It will not cause objects to be packed at the bit level.

2.12. PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

2.13. PASSIVE

The pragma has three forms :

```
PRAGMA PASSIVE;  
PRAGMA PASSIVE(SEMAPHORE);  
PRAGMA PASSIVE(INTERRUPT, <number>);
```

This pragma `Pragma passive` can be applied to a task or task type declared immediately within a library package spec or body. The pragma directs the compiler to optimize certain tasking operations. It is possible that the statements in a task body will prevent the intended optimization, in these cases a warning will be generated at compile time and will raise `TASKING_ERROR` at runtime.

2.14. PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

2.15. SHARED

This pragma is recognized by the implementation but has no effect.

2.16. STORAGE_UNIT

This pragma is recognized by the implementation. The implementation does not allow `SYSTEM` to be modified by means of pragmas, the `SYSTEM` package must be recompiled.

2.17. SUPPRESS

This pragma is implemented as described, except that `DIVISION_CHECK` and in some cases `OVERFLOW_CHECK` cannot be suppressed.

2.18. SYSTEM_NAME

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. Implementation-Dependent Attributes

3.1. P'REF

For a prefix that denotes an object, a program unit, a label, or an entry:

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type OPERAND defined in the package MACHINE_CODE. The attribute is only allowed within a machine code procedure.

See section F.4.8 for more information on the use of this attribute.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

3.2. T'TASKID

For a task object or a value T, T'TASK_ID yields the unique task id associated with a task. The value of this attribute is of the type ADDRESS in the package SYSTEM.

4. Specification Of Package SYSTEM

with UNSIGNED_TYPES;
package SYSTEM is

```
    pragma SUPPRESS(ALL_CHECKS);
    pragma SUPPRESS(EXCEPTION_TABLES);
    pragma NOT_ELABORATED;

    type NAME is ( doc_name );

    SYSTEM_NAME      : constant NAME := doc_name;

    STORAGE_UNIT     : constant := 0;
    MEMORY_SIZE      : constant := 16_777_216;

    -- System-Dependent Named Numbers

    MIN_INT          : constant := -2_147_483_648;
    MAX_INT          : constant := 2_147_483_647;
    MAX_DIGITS       : constant := 13;
    MAX_MANTISSA     : constant := 31;
    FINE_DELTA       : constant := 2.0**(-31);
    TICK             : constant := 0.01;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 99;

    MAX_REC_SIZE : integer := 64*1024;

    type ADDRESS is private;

    function ">" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "<" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function ">=" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "<=" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
    function "-" (A: ADDRESS; B: ADDRESS) return INTEGER;
    function "+" (A: ADDRESS; B: ADDRESS) return ADDRESS;
    function "*" (A: ADDRESS; I: INTEGER) return ADDRESS;
    function "/" (A: ADDRESS; I: INTEGER) return ADDRESS;

    function "<=" (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS;

    function MEMORY_ADDRESS
      (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS renames "<=";

    NO_ADDR : constant ADDRESS;
```

```

type TASK_ID is private;
NO_TASK_ID : constant TASK_ID;

type PROGRAM_ID is private;
NO_PROGRAM_ID : constant PROGRAM_ID;

private

type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_ADDR : constant ADDRESS := 0;

pragma BUILT_IN(">");
pragma BUILT_IN("<");
pragma BUILT_IN(">=");
pragma BUILT_IN("<=");
pragma BUILT_IN("-");
pragma BUILT_IN("+");

type TASK_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_TASK_ID : constant TASK_ID := 0;

type PROGRAM_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_PROGRAM_ID : constant PROGRAM_ID := 0;

end SYSTEM;

```

5. Restrictions On Representation Clauses

5.1. Pragma PACK

In the absence of pragma PACK record components are padded so as to provide for efficient access by the target hardware, pragma PACK applied to a record eliminate the padding where possible. Pragma PACK has no other effect on the storage allocated for record components a record representation is required.

5.2. Size Clauses

For scalar types a representation clause will pack to the number of bits required to represent the range of the subtype. A size clause applied to a record type will not cause packing of components; an explicit record representation clause must be given to specify the packing of the components. A size clause applied to a record type will cause packing of components only when the component type is a discrete type. An error will be issued if there is insufficient space allocated. The SIZE attribute is not supported for task, access, or floating point types.

5.3. Address Clauses

Address clauses are only supported for variables. Since default initialization of a variable requires evaluation of the variable address elaboration ordering requirements prohibit initialization of a variables which have address clauses. The specified address indicates the physical address associated with the variable.

5.4. Interrupts

Interrupt entries are not supported.

5.5. Representation Attributes

The ADDRESS attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

5.6. Machine Code Insertions

Machine code insertions are supported.

The general definition of the package `MACHINE_CODE` provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

```
CODE_n'( opcode, operand [, operand] );
```

where *n* indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

```
CODE_N'( opcode, (operand [, operand]) );
```

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

```
CODE_0'( op => opcode );
```

The *opcode* must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An *operand* can only be an entity defined in `MACHINE_CODE` or the 'REF attribute.

The arguments to any of the functions defined in `MACHINE_CODE` must be static expressions, string literals, or the functions defined in `MACHINE_CODE`. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

6. Conventions for Implementation-generated Names

There are no implementation-generated names.

7. Interpretation of Expressions in Address Clauses

Address expressions in an address clause are interpreted as physical addresses.

8. Restrictions on Unchecked Conversions

None.

9. Restrictions on Unchecked Deallocations

None.

10. Implementation Characteristics of I/O Packages

Instantiations of `DIRECT_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `DIRECT_IO` to provide an upper limit on the record size. In any case the maximum size supported is $1024 \times 1024 \times \text{STORAGE_UNIT}$ bits. `DIRECT_IO` will raise `USE_ERROR` if `MAX_REC_SIZE` exceeds this absolute limit.

Instantiations of `SEQUENTIAL_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE`'SIZE is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `INTEGER_IO` to provide an upper limit on the record size. `SEQUENTIAL_IO` imposes no limit on `MAX_REC_SIZE`.

11. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

11.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

11.2. Record and Array Sizes

The maximum size of a statically sized array type is $4,000,000 \times \text{STORAGE_UNITS}$. The maximum size of a statically sized record type is $4,000,000 \times \text{STORAGE_UNITS}$. A record type or array type declaration that exceeds these limits will generate a warning message.

11.3. Default Stack Size for Tasks

In the absence of an explicit `STORAGE_SIZE` length specification every task except the main program is allocated a fixed size stack of 10,240 `STORAGE_UNITS`. This is the value returned by `T'STORAGE_SIZE` for a task type `T`.

11.4. Default Collection Size

In the absence of an explicit `STORAGE_SIZE` length attribute the default collection size for an access type is 100 times the size of the designated type. This is the value returned by `T'STORAGE_SIZE` for an access type `T`.

11.5. Limit on Declared Objects

There is an absolute limit of $6,000,000 \times \text{STORAGE_UNITS}$ for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a **FATAL** error message.